Software Engineering II

Design, Models, Tools & Processes

Alan Blackwell

Objective of this course

- Software Engineering I
  - Understand the problems that you must solve (A)
- Programming in Java
  - Understand the technology you will use (B)
- Software Engineering II
  - Practical techniques for getting from A to B

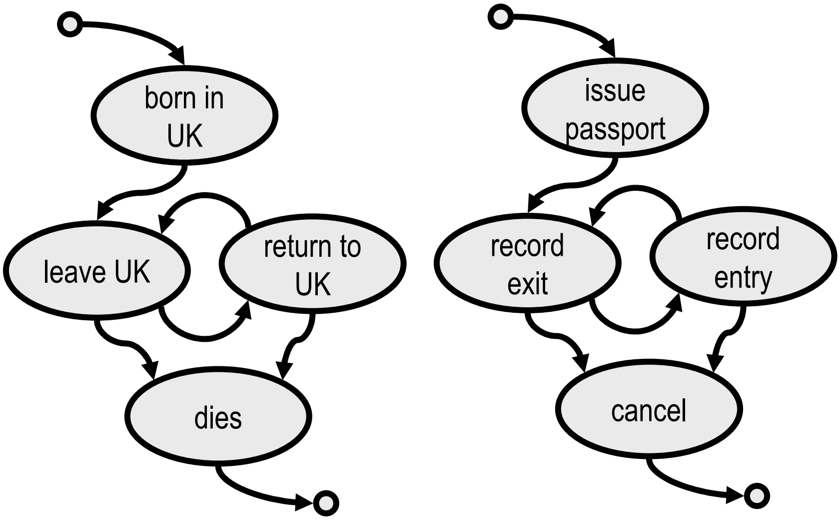
Where are the technical answers?

- Software Engineering I raised many problems
  - What technical means can we use to solve them?
- But "engineering, as it is properly understood, is not possible for software." (Paulson 2002)
- A systematic approach:
  - State what the system should do {D_1, D_2, D_3 …}
  - State what it shouldn't do {U_1, U_2, U_3 …}
  - Systematically add features that can be proven to implement D_n, while not implementing U_n
- Ideally a process of definition and proof

A candidate for definition and proof?

- The United Kingdom Passport Agency
  - http://www.parliament.the-stationery-office.co.uk/pa/cm199900/cmselect/cmpubacc/65/6509.htm
- 1997 contract for new computer system
  - aimed to improve issuing efficiency, on tight project timetable
  - project delays meant throughput not thoroughly tested
  - first live office failed the throughput criterion to continue roll-out
  - second office went live, roll out halted, but no contingency plan
  - rising backlog in early 1999, alongside increasing demand
  - passport processing times reached 50 days in July 1999
  - widespread publicity, anxiety and panic for travelling public
  - telephone service overloaded, public had to queue at UKPA offices
  - only emergency measures eventually reduced backlog
- So how hard can it be to issue a passport?
  - … let's try some simple definition

Defining a bureaucratic system

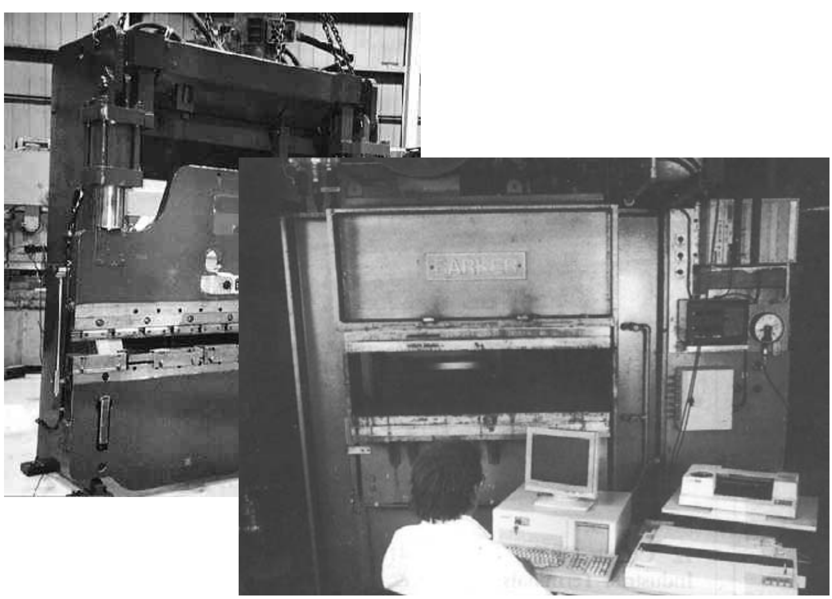


Defining a bureaucratic system

Why is the world complicated?

- Bureaucratic systems are complex because managers (and people) always mess up
  - Passports
  - Ambulance systems
  - University financials
- What about physical systems, which don’t rely on people to work?
  - Start with known characteristics of physical device.
  - Assemble behaviours to achieve function
  - This is how “real” engineering products (bridges and aircraft) are designed.

Defining a physical system



Defining a physical system

What is the problem?

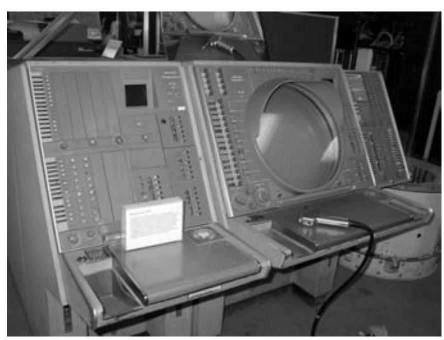
- The problem is ***not*** that we don't understand the computer.
- The problem ***is*** that we don't understand the problem!
- Does computer science offer any answers?
  - (or is this six hours of waffle?)
- The good news:
  - Computer scientists have been working on the problem of software engineering since 1968
- The bad news:
  - There is still no silver bullet!

Introduction

A design process based on knowledge

Pioneers – Bavarian Alps, 1968

- 1954: complexity of SAGE air-defence project was under-estimated by 6000 person-years …
  - … at a time when there were only about 1000 programmers in the whole world!
  - … “Software Crisis!”



- 1968: First meeting on “Software Engineering” convened in Garmisch-Partenkirchen.

Engineering and ignorance

- An engineer should be ignorant – systematically ignorant!
- Design is the process of learning about a problem and describing a solution
  - at first with many gaps …
  - eventually in sufficient detail to build it.

Learning by building models

- Software engineering is a process of gaining knowledge about a problem, and about its technical solution.
- We describe both the problem and the solution in a series of *design models*.
- Testing, manipulating and transforming those models helps us gather more knowledge.
- One of the most detailed models is written in a programming language.
  - Getting a working program is almost a side-effect of describing it!

Outline of course

- Roughly follows *Rational Unified Process*
- Inception
  - structured description of what system must do
- Elaboration
  - defining classes, data and system structure
- Construction
  - object interaction, behaviour and state
- Transition
  - testing and optimisation
- (phases may iterate in spiral/agile models)

Books

- ***Code Complete***: A practical handbook of software construction
  - Steve McConnell, Microsoft Press 1993
- ***UML Distilled*** (2nd edition)
  - Martin Fowler, Addison-Wesley 2000
- Further:
  - *Software Pioneers,* Broy & Denert
  - *Software Engineering,* Roger Pressman
  - *Interaction Design,* Holtzblatt & Meyer
  - *The Sciences of the Artificial*, Herb Simon
  - *Educating the Reflective Practitioner*, Donald Schon
  - *The Golem at Large,* Collins & Pinch
  - *To Engineer is Human,* Petroski
  - *What Engineers Know,* Vincenti

Exam questions

- This course has been completely revised for 2004/05
  - Some past exam questions cover material that is no longer in the course (especially Z and ML).
- There is substantial new material
  - Some has been taught elsewhere in the Tripos (especially UML):
  - Programming in Java 2004, Paper 1, Q10
  - Software Engineering and Design 2003 Paper 10, Q12 and 2004 Paper 11, Q11
  - Additional Topics 2000, Paper 7, Q13

Unified Modeling Language

- Combines several methods from early 90s
  - Grady Booch “Rational Rose”
  - Ivar Jacobson “Object Oriented Software Engineering”
  - Jim Rumbaugh “Object Modeling Technique”
- Competition and consolidation
  - Natural leaders emerged, mostly by merit
  - Jacobson joined Rational Software
  - Object Management Group blessed the result
  - UML v1.1 1997, v1.3 1998

UNIFIED MODELING LANGUAGE

Using a design language

- “Organic” hacking doesn’t work when:
  - Many programmers on project.
  - Too large to hold in your head.
  - Need for accurate estimates.
  - Several companies involved.
- So design techniques must provide:
  - language for communication
  - decomposition and simplification
  - predictable relationship to implementation language
  - basis for contractual agreements

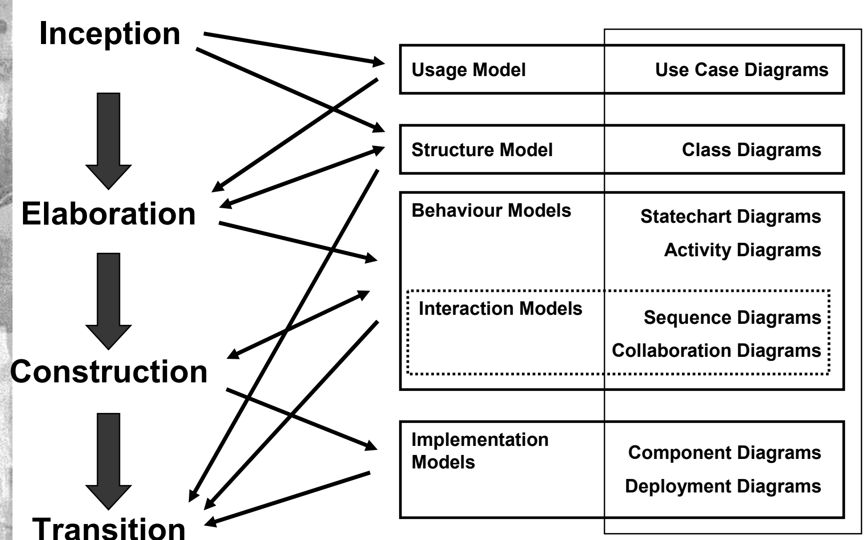
Modeling (“CASE”) tools

- Computer Aided Software Engineering
- Diagram editors: enforce syntax
  - drawing packages or specialist diagram tools will do
- Repositories: understand diagram content
  - maintain name/type database, diagram consistency
- Code generation: at the least, saves typing
  - dumping class signatures in Java/C++ syntax is easy
  - anything more is hard (and perhaps pointless)
- Use with non-OO languages
  - inheritance, instantiation can be implemented in C etc.
  - OO design can be exploited in later development work

UML diagrams - overview

- **Use Case** diagrams - interactions with / interfaces to the system.
- **Class** diagrams - type structure of the system.
- **Collaboration** diagrams - interaction between instances
- **Sequence** diagrams - temporal structure of interaction
- **Activity** diagrams - ordering of operations
- **Statechart** diagrams - behaviour of individual objects
- **Component** and **Deployment** diagrams - system organisation

Overview of a design process with UML



Inception

structured description of system usage and function

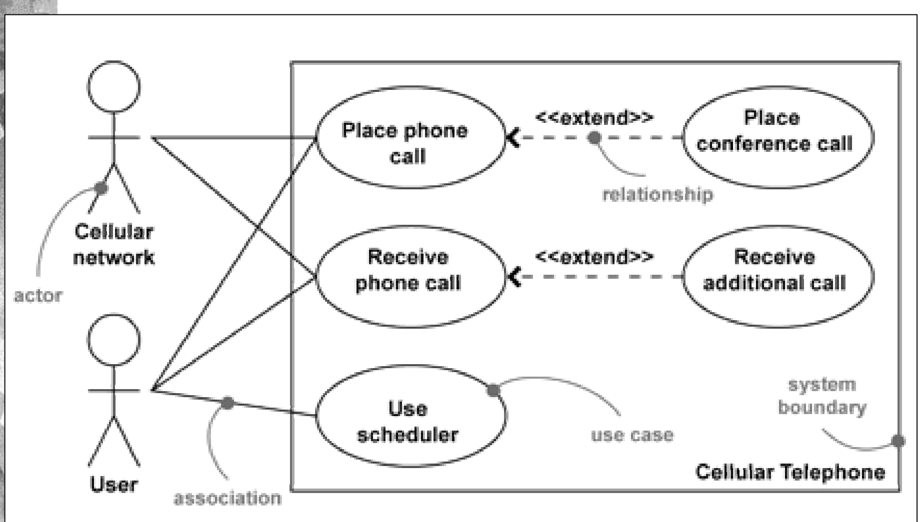
Pioneers – Tom DeMarco

- Structured Analysis
  - 1978, Yourdon Inc
- Defined the critical technical role of the system analyst
  - Analyst acts as a middleman between user and developer
- Analyst's job is to construct a functional specification
  - data dictionary
  - data flow
  - partitioning

Analysis scenarios

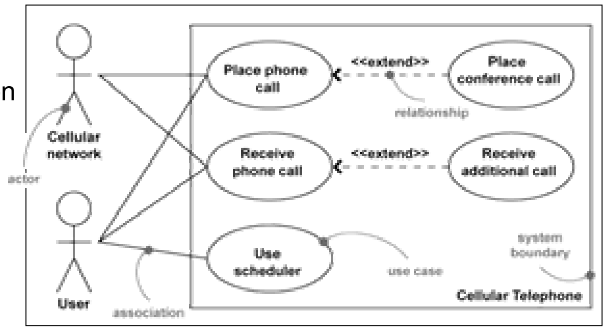
- Describe the human activity that the system has to carry out or support.
  - Known as *use cases* in UML
- Used to discover and record object interactions (collaborations).
- Can be developed as a group activity.
- May be based on observation & interview techniques such as *Contextual Inquiry* (Beyer & Holtzblatt 1997)

UML Use Case diagram



UML Use Case diagram

- Actors
  - play system *role*
  - may not be people
- Use case
  - like a scenario
- Relationships
  - include
  - extend
  - generalisation



Objects in a scenario

- The ***nouns*** in a description refer to 'things'.
  - A source of classes and objects.
- The ***verbs*** refer to actions.
  - A source of interactions between objects.
  - Actions describe object behavior, and hence required methods.

Example of problem description

The cinema booking system should store seat bookings for multiple theatres.
Each theatre has seats arranged in rows.
Customers can reserve seats and are given a row number and seat number.
They may request bookings of several adjoining seats.
Each booking is for a particular show (i.e., the screening of a given movie at a certain time).
Shows are at an assigned date and time, and scheduled in a theatre where they are screened.
The system stores the customers' telephone number.

Extracted nouns & verbs

Cinema booking system
Stores (seat bookings)
Stores (telephone number)

Theatre
Has (seats)

Movie

Customer
Reserves (seats)
Is given (row number, seat number)
Requests (seat booking)

Time

Date

Seat booking

Show
Is scheduled (in theatre)

Seat

Seat number

Telephone number

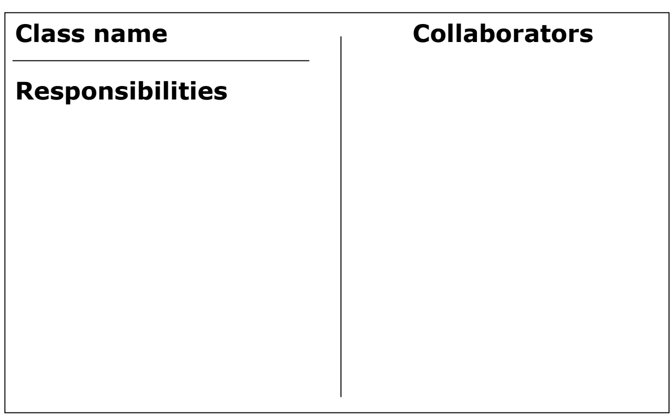
Row

Row number

Scenario structure: CRC cards

- First described by Kent Beck and Ward Cunningham.
  - Later innovators of "agile" programming (more on this later in course)
- Use simple index cards, with each cards recording:
  - A *class* name.
  - The class's *responsibilities*.
  - The class's *collaborators*.

Typical CRC card



Partial example

`CinemaBookingSystem`	*Collaborators*
Can find shows by title and day.	**Show**
Stores collection of shows.	**Collection**
Retrieves and displays show details.	
...	

Refinement of usage model

- Scenarios allow you to check that the problem description is clear and complete.
- Analysis leads gradually into design.
  - Talking through scenarios & class responsibilities leads to elaborated models.
- Spotting errors or omissions here will save considerable wasted effort later!
  - Sufficient time should be taken over the analysis.

Elaboration

defining classes, data and system structure

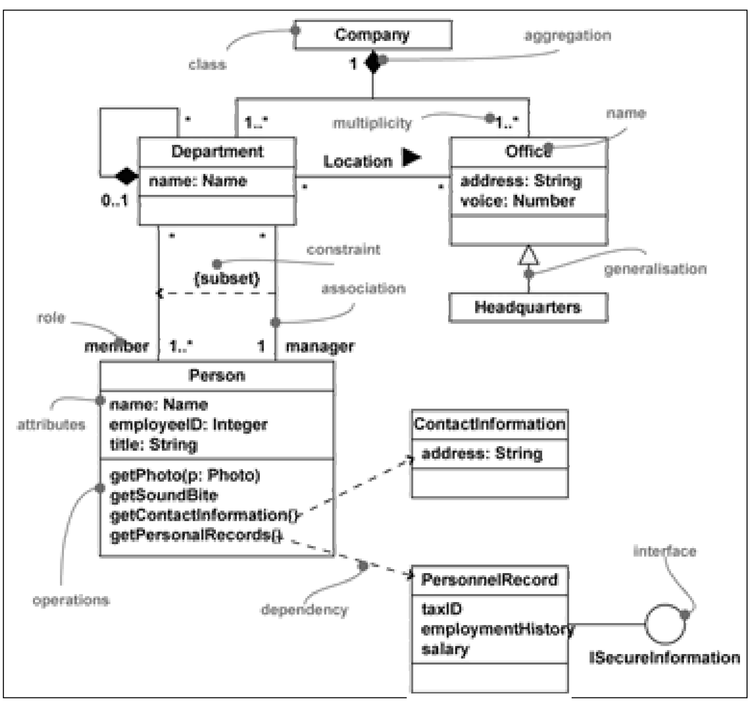
Pioneers – Peter Chen

- Entity-Relationship Modeling
  - 1976, Massachusetts Institute of Technology
- User-oriented response to Codd's relational database model
  - Define attributes and values
  - Relations as associations between things
  - Things play a *role* in the relation.
- Diagrams show entity (box), relation (diamond), role (links).

Review of objects and classes

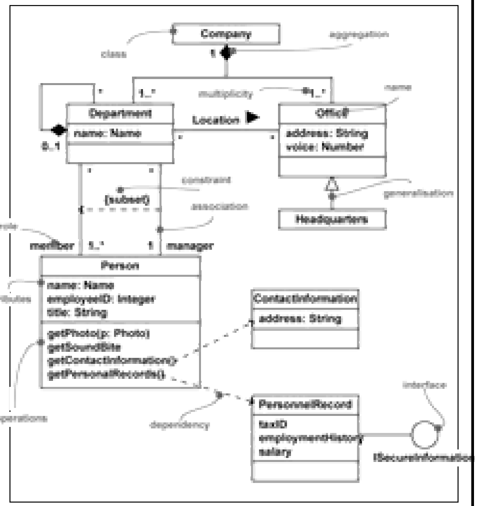
- objects
  - represent ‘things’ in some problem domain (example: “the red car down in the car park”)
- classes
  - represent all objects of a kind (example: “car”)
- operations
  - actions invoked on objects (Java “*methods”)*
- instance
  - can create many instances from a single class
- state
  - all the attributes (field values) of an instance

UML Class diagram



UML Class diagram

- Attributes
  - *type* and *visibility*
- Operations
  - signature and visibility
- Relationships
  - *association*
    - with *multiplicity*
    - potentially *aggregation*
  - *generalisation*



Class design from CRC cards

- Scenario analysis helps to clarify application structure.
  - Each card maps to a class.
  - Collaborations reveal class cooperation/object interaction.
- Responsibilities reveal public methods.
  - And sometimes fields; e.g. “Stores collection ...”

Refining class interfaces

- Replay the scenarios in terms of method calls, parameters and return values.
- Note down the resulting method signatures.
- Create outline classes with public-method stubs.
- Careful design is a key to successful implementation.

Dividing up a design model

- Abstraction
  - Ignore details in order to focus on higher level problems (e.g. aggregation, inheritance).
  - If classes correspond well to types in domain they will be easy to understand, maintain and reuse.
- Modularization
  - Divide model into parts that can be built and tested separately, interacting in well-defined ways.
  - Allows different teams to work on each part
  - Clearly defined interfaces mean teams can work independently & concurrently, with increased chance of successful integration.

Implementing multiple associations

- Most applications involve collections of objects
  - java.util package contains classes for this
- The number of items to be stored varies
  - Items can be added and deleted
  - Collection increases capacity as necessary
  - Count of items obtained with size()
  - Items kept in order, accessed with *iterator*
- Details of how all this is done are hidden.

Pioneers – David Parnas

- Information Hiding
  - 1972, Carnegie Mellon University
- How do you decide the points at which a program should be split into pieces?
  - Are small modules better?
  - Are big modules better?
  - What is the optimum boundary size?
- Parnas proposed the best criterion for modularization:
  - Aim to hide design decisions within the module.

Information hiding in OO models

- Data belonging to one object is hidden from other objects.
  - Know *what* an object can do, not *how* it does it.
  - Increases independence, essential for large systems and later maintenance
- Use visibility to hide implementation
  - Only methods intended for interface to other classes should be public.
  - Fields should be private – accessible only within the same class.
  - *Accessor* methods provide information about object state, but don’t change it.
  - *Mutator* methods change an object’s state.

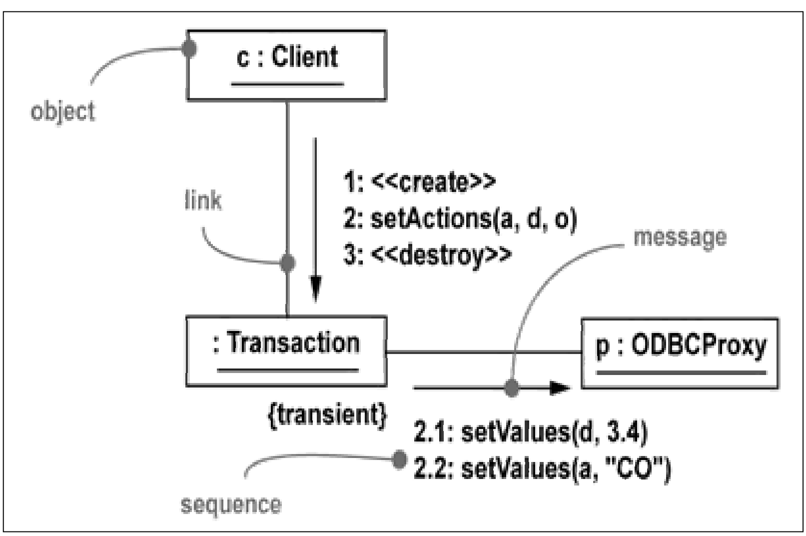
Cohesion in OO models

- Aim for high cohesion:
  - Each component achieves only “one thing”
- Method (functional) cohesion
  - Method only performs out one operation
  - Groups things that must be done together
- Class (type) cohesion
  - Easy to understand & reuse as a domain concept
- Causes of low, poor, cohesion
  - Sequence of operations with no necessary relation
  - Unrelated operations selected by control flags
  - No relation at all – just a bag of code

Construction

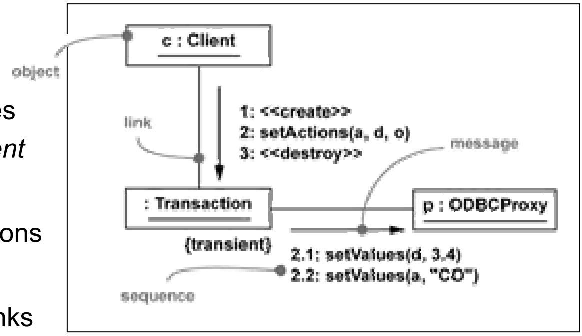
object interaction, behaviour and state

UML Collaboration diagram



UML Collaboration diagram

- Objects
  - class instances
  - can be *transient*
- Links
  - from associations
- Messages
  - travel along links
  - numbered to show *sequence*



Loose coupling

- Coupling: links between parts of a program.
- If two classes depend closely on details of each other, they are *tightly coupled*.
- We aim for *loose coupling*.
  - keep parts of design clear & independent
  - may take several design iterations
- Loose coupling makes it possible to:
  - achieve reusability, modifiability
  - understand one class without reading others;
  - change one class without affecting others.
- Thus improves maintainability.

Responsibility-driven design

- Which class should I add a new method to?
  - Each class should be responsible for manipulating its own data.
  - The class that owns the data should be responsible for processing it.
- Leads to low coupling & “client-server contracts”
  - Consider every object as a server
  - Improves reliability, partitioning, graceful degradation

Interfaces as specifications

- Define method *signatures* for classes to interact
  - Include parameter and return types.
  - Strong separation of required functionality from the code that implements it (information hiding).
- Clients interact independently of the implementation.
  - But clients can choose from alternative implementations.

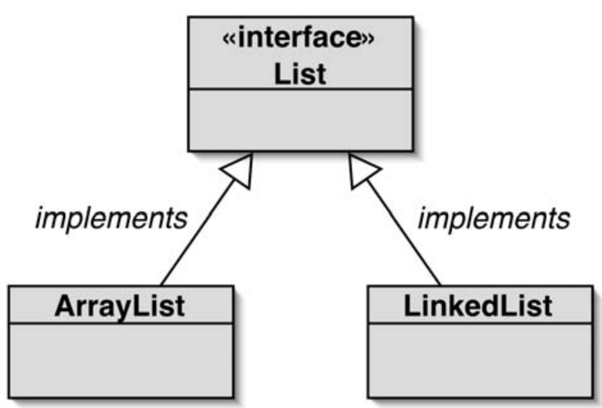
Interfaces in Java

- Provide specification without implementation.
  - Fully abstract – implementing classes don't inherit code
  - Support not only polymorphism, but *multiple inheritance*
  - implementing classes are still subtypes of the interface type, but allowed more than one "parent".

```
public class Fox extends Animal implements Drawable

public class Hunter implements Actor, Drawable
```

Alternative implementations



Causes of error situations

- Incorrect implementation.
  - Does not meet the specification.
- Inappropriate object request.
  - E.g., invalid index.
- Inconsistent or inappropriate object state.
  - E.g. arising through class extension.
- Not always programmer error
  - Errors often arise from the environment (incorrect URL entered, network interruption).
  - File processing often error-prone (missing files, lack of appropriate permissions).

Defensive programming

- Client-server interaction.
  - Should a server assume that clients are well-behaved?
  - Or should it assume that clients are potentially hostile?
- Significant differences in implementation required.
- Issues to be addressed
  - How much checking by a server on method calls?
  - How to report errors?
  - How can a client anticipate failure?
  - How should a client deal with failure?

Argument values

- Arguments represent a major ‘vulnerability’ for a server object.
  - Constructor arguments initialize state.
  - Method arguments often control behavior.
- Argument checking is one defensive measure.
- How to report illegal arguments?
  - To the user? (Is there a human user? Can they solve the problem?)
  - To the client object: return a diagnostic value, or throw an exception.

Example of diagnostic return

```
public boolean removeDetails(String key)
{
    if(keyInUse(key)) {
        ContactDetails details =
                       (ContactDetails) book.get(key);
        book.remove(details.getName());
        book.remove(details.getPhone());
        numberOfEntries--;
        return true;
    }
    else {
        return false;
    }
}
```

Client response to diagnostic

- Test the return value.
  - Attempt recovery on error.
  - Avoid program failure.
- Ignore the return value.
  - Cannot be prevented.
  - Likely to lead to program failure.
- Exceptions are preferable.

Exception-throwing

- Special language feature
  - Java does provide exceptions
- Advantages
  - No ‘special’ return value needed.
  - Errors cannot be ignored in the client.
- Disadvantages (or are they?)
  - The normal flow-of-control is interrupted.
  - Specific recovery actions are encouraged.

Example of argument exception

```
public ContactDetails getDetails(String key)
{
    if(key == null) {
        throw new NullPointerException(
                        "null key in getDetails");
    }
    if(key.trim().length() == 0) {
        throw new IllegalArgumentException(
                "Empty key passed to getDetails");
    }
    return (ContactDetails) book.get(key);
}
```

Error response and recovery

- Clients should take note of error notifications.
  - Check return values.
  - Don’t ‘ignore’ exceptions.
- Include code to attempt recovery.
  - Will often require a loop.

Attempting recovery

```
// Try to save the address book.
boolean successful = false;
int attempts = 0;
do {
    try {
        addressbook.saveToFile(filename);
        successful = true;
    }
    catch(IOException e) {
        System.out.println("Unable to save to " + filename);
        attempts++;
        if(attempts < MAX_ATTEMPTS) {
            filename = an alternative file name;
        }
    }
} while(!successful && attempts < MAX_ATTEMPTS);
if(!successful) {
    Report the problem and give up;
}
```

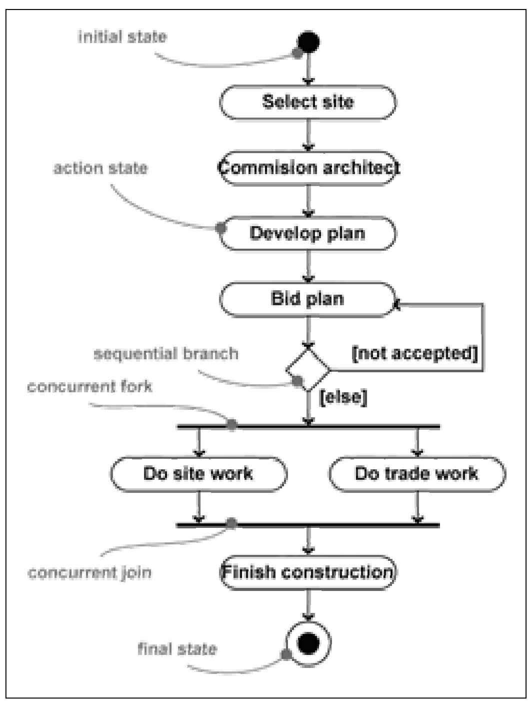
Error avoidance

- Clients can often use server query methods to avoid errors.
  - More robust clients mean servers can be more trusting.
  - Unchecked exceptions can be used.
  - Simplifies client logic.
- May increase client-server coupling.

More construction

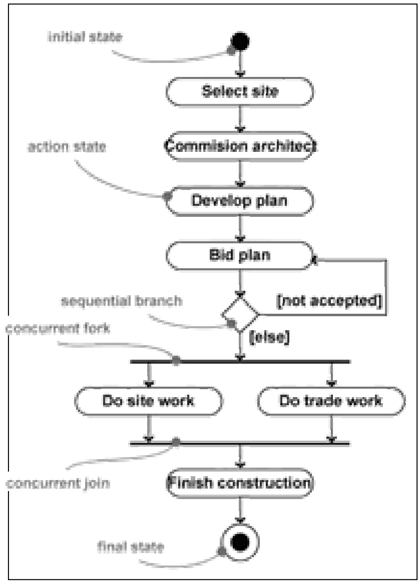
object internals

UML Activity diagram



UML Activity diagram

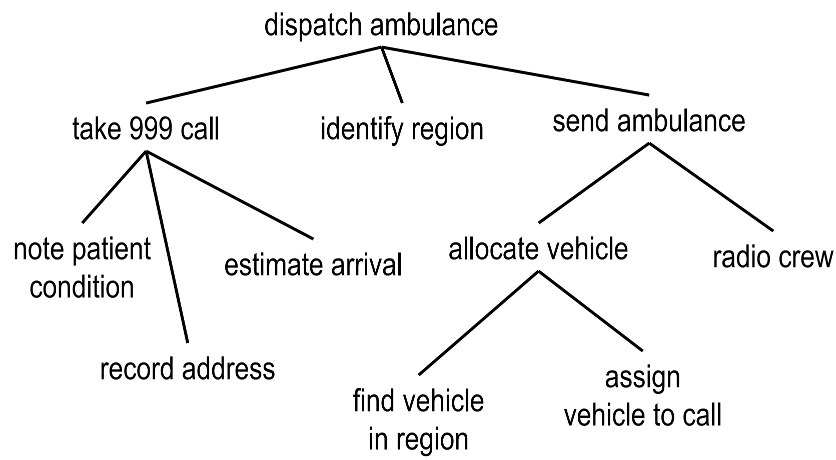
- Like flow charts
  - Activity as action states
- Flow of control
  - transitions
  - branch points
  - concurrency (fork & join)
- Illustrate flow of control
  - high level - e.g. workflow
  - low level - e.g. lines of code



Pioneers – Edsger Dijkstra

- Structured Programming
  - 1968, Eindhoven
- Why are programmers so bad at understanding dynamic processes and concurrency?
  - (ALGOL then – but still hard in Java today!)
- Observed that “go to” made things worse
  - Hard to describe what state a process has reached, when you don’t know which process is being executed.
- Define process as nested set of execution blocks, with fixed entry and exit points

Top-down design & stepwise refinement



Bottom-up construction

- Why?
  - Start with what you understand
  - Build complex structures from well-understood parts
  - Deal with concrete cases in order to understand abstractions
- Real design combines top-down and bottom up.

Modularity at code level

- Is this “routine” required?
- Define what it will do
  - What information will it hide?
  - Inputs
  - Outputs (including side effects)
  - How will it handle errors?
- Give it a good name
- How will you test it?
- Think about efficiency and algorithms
- Write as comments, then fill in actual code

Modularity in non-OO languages

- Separate source files in C
  - Inputs, outputs, types and interface functions defined by declarations in “header files”.
  - Private variables and implementation details defined in the “source file”
- Modules in ML, Perl, Fortran, …
  - Export publicly visible interface details.
  - Keep implementation local whenever possible, in interest of information hiding, encapsulation, low coupling.

Source code as a design model

- Objectives:
  - Accurately express logical structure of the code
  - Consistently express the logical structure
  - Improve readability
- Good visual layout shows program structure
  - Mostly based on white space and alignment
  - The compiler ignores white space
  - Alignment is the single most obvious feature to human readers.
- Code layout is most like the art of typography

Code as a structured model

```
Function_name (parameter1, parameter2)

// Function which doesn't do anything, beyond showing the fact
// that different parts of the function can be distinguished.

    type1: local_data_A, local_data_B
    type2: local_data_C

    // Initialisation section
    local_data_A := parameter1 + parameter2;
    local_data_B := parameter1 - parameter2;
    local_data_C := 1;

    // Processing
    while (local_data_C < 40) {
        if ( (local_data_B ^ 2) > local_data_A ) then {
            local_data_B := local_data_B - 1;
        } else {
            local_data_B := local_data_B + 1;
        } // end if
        local_data_C := local_data_C + 1;
    } // end while

} // end function
```

Expressing local control structure

```
while (local_data_C < 40) {
    form_initial_estimate(local_data_C);
    record_marker(local_data_B – 1);
    refine_estimate(local_data_A);
    local_data_C := local_data_C + 1;
} // end while
```

```
if ( (local_data_B ^ 2) > local_data_A ) then {
    // drop estimate
    local_data_B := local_data_B – 1;
} else {
    // raise estimate
    local_data_B := local_data_B + 1;
} // end if
```

Expressing structure within a line

- Whitespacealwayshelpshumanreaders
  - `newtotal=oldtotal+increment/missamount-1;`
  - `newtotal = oldtotal + increment / missamount - 1;`
- The compiler doesn't care – take care!
  - `x = 1  *  y+2  *  z;`
- Be conservative when nesting parentheses
  - `while ( (! error) && readInput() )`
- Continuation lines – exploit alignment

```
if ( ( aLongVariableName & anotherLongOne ) |
     ( someOtherCondition() ) )
   {
    …
   }
```

Naming variables: Form

- Priority: full and accurate (*not* just short)
  - Abbreviate for pronunciation (remove vowels)
    - e.g. CmptrScnce (leave first and last letters)
- Parts of names reflect conventional functions
  - Role in program (e.g. “count”)
  - Type of operations (e.g. “window” or “pointer”)
  - Hungarian naming (not really recommended):
    - e.g. pscrMenu, ichMin
- Even individual variable names can exploit typographic structure for clarity
  - **`xPageStartPosition`**
  - **`x_page_start_position`**

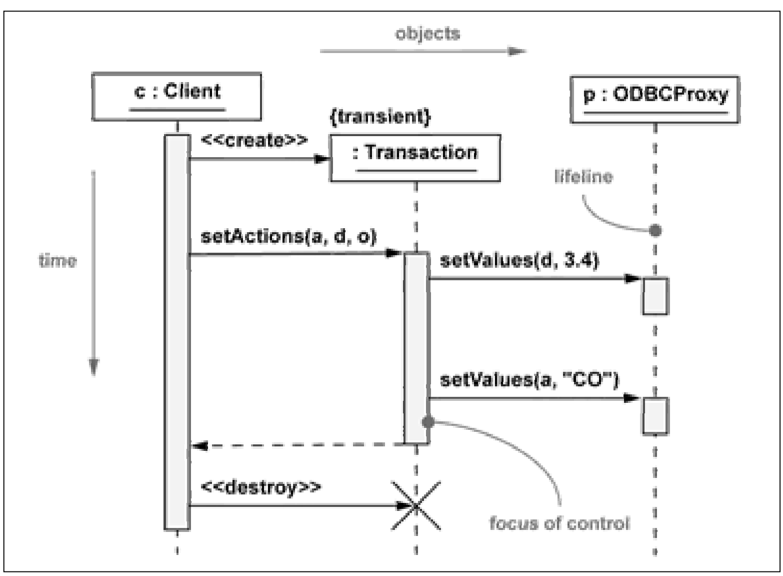
Naming variables: Content

- Data names describe domain, not computer
  - Describe what, not just how
  - **`CustomerName`** better than **`PrimaryIndex`**
- Booleans should have obvious truth values
  - **`ErrorFound`** better than **`Status`**
- Indicate which variables are related
  - **`CustName`**, **`CustAddress`**, **`CustPhone`**
- Identify globals, types & constants (in C)
  - e.g. g_wholeApplet, T_mousePos
- Even temporary variables have meaning
  - **`Index`**, not **`Foo`**

More construction

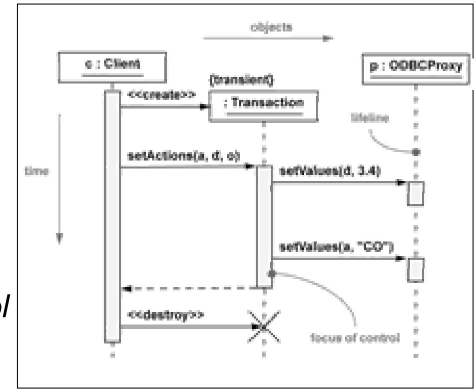
object interaction & data processing

UML Sequence diagram



UML Sequence diagram

- Interaction again
  - same content as collaboration
  - emphasises time dimension
- Object *lifeline*
  - objects across page
  - time down page
- Shows *focus of control*



Pioneers – Michael Jackson

- Jackson Structured Programming
  - 1975, independent consultant, London
- Describe program structure according to the structure of input and output streams
  - Mostly used for COBOL file processing
  - Still relevant to stream processing in Perl

Data structure vs. code structure

- Data records (items in collection, elements in array) require a code loop
- Variant cases (subtypes, categories, enumerations) require conditional execution
- Switching between code and data perspectives helps to learn about design complexity and to check correctness.
  - “Guard” is a data condition that should hold before entering some section of code.
  - “Invariant” is a data condition that should be the same both before and after some section of code.
    - (ideas originally from Hoare – see later)

Data in a while loop

```
checkDataOrder();

while ( dataIncomplete() ) {
  reorderDataStructure();
  restoreDataOrder();
}

checkDataOrder();
```

Invariant holds here

Guard holds here

Restore invariant

Invariant and ***not*** guard holds here

Structural *roles* of variables

- Classification of what variables do in a routine
  - Don’t confuse with data types (e.g. int, char, float)
- Almost all variables in simple programs do one of:
  - fixed value
  - stepper
  - most-recent holder
  - most-wanted holder
  - gatherer
  - transformation
  - one-way flag
  - follower
  - temporary
  - organizer
- Most common (70 % of variables) are fixed value, stepper or most-recent holder.

Fixed value

- Value is never changed after initialization
- Example: input radius of a circle, then print area
  - variable r is a *fixed value,* gets its value once, never changes after that.
- Useful to declare “final” in Java.

```
public class AreaOfCircle {

  public static void main(String[] args) {
      final float PII = 3.14F;
      float r;
      System.out.print("Enter circle radius: ");
      r = UserInputReader.readFloat();
      System.out.println(“Circle area is " + PII * r * r);
  }
}
```

Stepper

- Goes through a succession of values in some systematic way
  - E.g. counting items, moving through array index
- Example: loop where multiplier is used as a stepper.
  - outputs multiplication table, stepper goes through values from one to ten.

```
public class MultiplicationTable {

  public static void main(String[] args) {
      int multiplier;
      for (multiplier = 1; multiplier <= 10; multiplier++)
             System.out.println(multiplier + " * 3 = "
                    + multiplier * 3);
  }
}
```

Most-recent holder

- Most recent member of a group, or simply latest input value
- Example: ask the user for input until valid.
  - Variable s is a most-recent holder since it holds the latest input value.

```
public class AreaOfSquare {

  public static void main(String[] args) {
      float s = 0f;
      while (s <= 0) {
             System.out.print("Enter side of square: ");
             s = UserInputReader.readFloat();
      }
      System.out.println("Area of square is " + s * s);
  }
}
```

Most-wanted holder

- The "best" (biggest, smallest, closest) of values seen.
- Example: find smallest of ten integers.
  - Variable smallest is a *most-wanted holder* since it is given the most recent value if it is smaller than the smallest one so far.
  - (i is a *stepper* and number is a *most-recent holder*.)

```
public class SearchSmallest {
   public static void main(String[] args) {
        int i, smallest, number;
        System.out.print("Enter the 1. number: ");
        smallest = UserInputReader.readInt();
        for (i = 2; i <= 10; i++) {
                System.out.print("Enter the " + i + ". number: ");
                number = UserInputReader.readInt();
                if (number < smallest) smallest = number;
        }
        System.out.println("The smallest was " + smallest);
   }
}
```

Gatherer

- Accumulates values seen so far.
- Example: accepts integers, then calculates mean.
  - Variable sum is a *gatherer* the total of the inputs is gathered in it.
  - (count is a *stepper* and number is a *most-recent holder*.)

```
public class MeanValue {

   public static void main(String[] argv) {
        int count=0;
        float sum=0, number=0;
        while (number != -999) {
                System.out.print("Enter a number, -999 to quit: ");
                number = UserInputReader.readFloat();
                if (number != -999) { sum += number; count++; }
        }
        if (count>0) System.out.println("The mean is " +
                                        sum / count);
   }
}
```

Transformation

- Gets every value by calculation from the value of other variable(s).
- Example: ask the user for capital amount, calculate interest and total capital for ten years.
  - Variable interest is a *transformation* and is always calculated from the capital.
  - (capital is a *gatherer* and i is a *counter*.)

```
public class Growth {
   public static void main(String[] args) {
        float capital, interest;   int i;
        System.out.print("Enter capital (positive or negative): ");
        capital = UserInputReader.readFloat();
        for (i = 1; i <=10; i++) {
                interest = 0.05F * capital;
                capital += interest;
                System.out.println("After "+i+" years interest is "
                        + interest + " and capital is " + capital);
        }
   }
}
```

One-way flag

- Boolean variable which, once changed, never returns to its original value.
- Example: sum input numbers and report if any negatives.
  - The *one-way flag* neg monitors whether there are negative numbers among the inputs. If a negative value is found, it will never return to false.
  - (number is a *most-recent holder* and sum is a *gatherer*.)

```
public class SumTotal {
   public static void main(String[] argv) {
        int number=1, sum=0;
        boolean neg = false;
        while (number != 0) {
                System.out.print("Enter a number, 0 to quit: ");
                number = UserInputReader.readInt();  sum += number;
                if (number < 0) neg = true;
        }
        System.out.println("The sum is " + sum);
        if (neg) System.out.println("There were negative numbers.");
   }
}
```

Follower

- Gets old value of another variable as its new value.
- Example: input twelve integers and find biggest difference between successive inputs.
  - Variable previous is a *follower*, following current.

```
public class BiggestDifference {
   public static void main(String[] args) {
      int month, current, previous, biggestDiff;
      System.out.print("1st: "); previous = UserInputReader.readInt();
      System.out.print("2nd: "); current = UserInputReader.readInt();
      biggestDiff = current - previous;
      for (month = 3; month <= 12; month++) {
        previous = current;
        System.out.print(month + “th: ");
        current = UserInputReader.readInt();
        if (current - previous > biggestDiff)
          biggestDiff = current - previous;
      }
      System.out.println(“Biggest difference was " + biggestDiff);
   }
}
```

Temporary

- Needed only for very short period (e.g. between two lines).
- Example: output two numbers in size order, swapping if necessary.
  - Values are swapped using a *temporary* variable tmp whose value is later meaningless (no matter how long the program would run).

```
public class Swap {
   public static void main(String[] args) {
        int number1, number2, tmp;
        System.out.print("Enter num: ");
        number1 = UserInputReader.readInt();
        System.out.print("Enter num: ");
        number2 = UserInputReader.readInt();
        if (number1 > number2) {
                tmp = number1;
                number1 = number2;
                number2 = tmp;
        }
        System.out.println(“Order is " + number1 + “," + number2 + ".");
   }
}
```

Organizer

- An array for rearranging elements
- Example: input ten characters and output in reverse order.
  - The reversal is performed in *organizer* variable word.
  - tmp is a *temporary* and i is a *stepper*.)

```
public class Reverse {
   public static void main(String[] args) {
        char[] word = new char[10];
        char tmp; int i;
        System.out.print("Enter ten letters: ");
        for (i = 0; i < 10; i++) word[i] = UserInputReader.readChar();
        for (i = 0; i < 5; i++) {
                tmp = word[i];
                word[i] = word[9-i];
                word[9-i] = tmp;
        }
        for (i = 0; i < 10; i++) System.out.print(word[i]);
        System.out.println();
   }
}
```

Verifying variables by role

- Many student program errors result from swapping variables between roles.
  - Identify role of each variable during design
- Many opportunities to check correct operation according to constraints on role
  - Check stepper within range
  - Check most-wanted meets selection criterion
  - De-allocate temporary value
  - Confirm size of organizer array is invariant
  - Use compiler to guarantee final fixed value
- Either do runtime safety checks (noting efficiency tradeoff), or use language features.

Type-checking as modeling tool

- Refine types to reflect meaning, not just to satisfy the compiler.
- Valid (to compiler), but incorrect, code:
  - `float totalHeight, myHeight, yourHeight;`
  - `float totalWeight, myWeight, yourWeight;`
  - `totalHeight = myHeight + yourHeight + myWeight;`
- Type-safe version:
  - `type t_height, t_weight: float;`
  - `t_height totalHeight, myHeight, yourHeight;`
  - `t_weight totalWeight, myWeight, yourWeight;`
  - `totalHeight = myHeight + yourHeight + myWeight;`

Compile error!

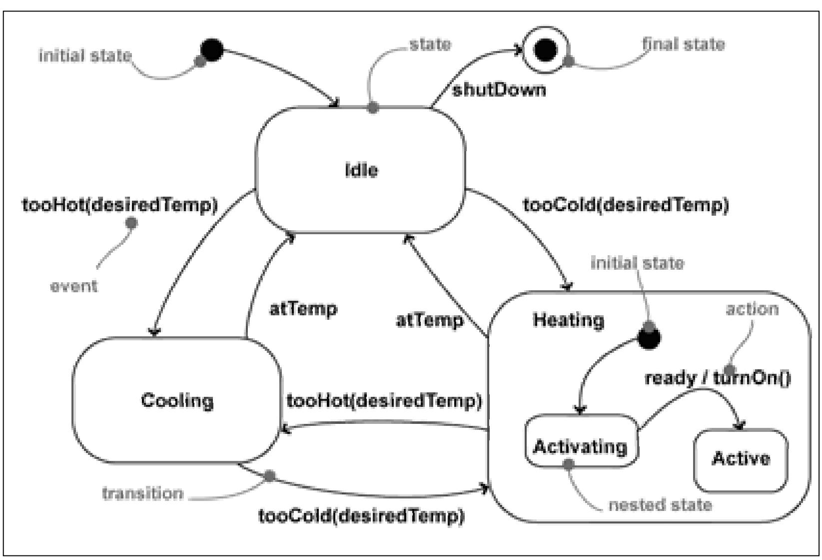
Language support for user types

- Smalltalk
  - All types are classes – consistent, but inefficient
- C++
  - Class overhead very low
  - User-defined types have no runtime cost
- Java
  - Unfortunately a little inefficient
  - But runtime inefficiency in infrequent calculations far better than lost development time.

More construction

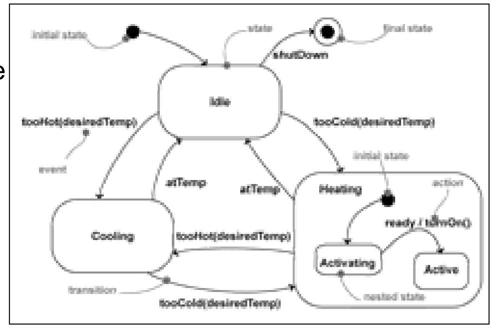
object state

UML Statechart diagram



UML Statechart diagram

- Object lifecycle
  - data as state machine
- Harel statecharts
  - nested states
  - concurrent substates
- Explicit initial/final
  - valuable in C++
- Note inversion of activity diagram



Maintaining valid system state

- Pioneers (e.g. Turing) talked of proving program correctness using mathematics
- In practice, the best we can do is confirm that the state of the system is consistent
  - State of an object valid before and after operation
  - Parameters and local variables valid at start and end of routine
  - Guard values define state on entering & leaving control blocks (loops and conditionals)
  - Invariants define conditions to be maintained throughout operations, routines, loops.

Pioneers – Tony Hoare

- Assertions and proof
  - 1969, Queen's University Belfast
- Program element behaviour can be defined
  - by a *post-condition* that will result …
  - … given a known *pre-condition*.
- If prior and next states accurately defined:
  - Individual elements can be composed
  - Program correctness is potentially provable

Formal models: Z notation

$$
\begin{array}{|l}
\hline
\textit{BirthdayBook} \\
\textit{known} : \mathbb{P}\,\textit{NAME} \\
\textit{birthday} : \textit{NAME} \rightarrow\!\!\!\!\!\!+\; \textit{DATE} \\
\hline
\textit{known} = \operatorname{dom} \textit{birthday} \\
\hline
\end{array}
$$

- Definitions of the *BirthdayBook* state space:
  - *known* is a set of NAMEs
  - *birthday* is a partial map from NAMEs to DATEs
- Invariants:
  - *known* must be the domain of *birthday*

Formal models: Z notation

$$
\begin{array}{|l}
\hline
\textit{AddBirthday} \\
\Delta \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\hline
\textit{name}? \notin \textit{known} \\
\textit{birthday}' = \textit{birthday} \cup \{\textit{name}? \mapsto \textit{date}?\} \\
\hline
\end{array}
$$

- An operation to change state
  - *AddBirthday* modifies the state of *BirthdayBook*
  - Inputs are a new *name* and *date*
  - Precondition is that *name* must not be previously known
  - Result of the operation, *birthday’* is defined to be a new and enlarged domain of the *birthday* map function

Formal models: Z notation

$$
\begin{array}{|l}
\hline
\textit{Remind} \\
\Xi \textit{BirthdayBook} \\
\textit{today}? : \textit{DATE} \\
\textit{cards}! : \mathbb{P}\, \textit{NAME} \\
\hline
\textit{cards}! = \{ n : \textit{known} \mid \textit{birthday}(n) = \textit{today}? \} \\
\hline
\end{array}
$$

- An operation to inspect state of *BirthdayBook*
  - This schema does not change the state of *BirthdayBook*
  - It has an output value (a set of people to send *cards* to)
  - The output set is defined to be those people whose birthday is equal to the input value *today*.

Advantages of formal models

- Requirements can be analysed at a fine level of detail.
- They are declarative (specify what the code should do, not how), so can be used to check specifications from an alternative perspective.
- As a mathematical notation, offer the promise of tools to do automated checking, or even proofs of correctness (“verification”).
- They have been applied in some real development projects.

Disadvantages of formal models

- Notations that have lots of Greek letters and other weird symbols look scary to non-specialists.
  - Not a good choice for communicating with clients, users, rank-and-file programmers and testers.
- Level of detail (and thinking effort) is similar to that of code, so managers get impatient.
  - If we are working so hard, why aren’t we just writing the code?
- Tools are available, but not hugely popular.
  - Applications so far in research / defence / safety critical
- Pragmatic compromise from UML developers
  - “Object Constraint Language” (OCL).
  - Formal specification of some aspects of the design, so that preconditions, invariants etc. can be added to models.

Language support for assertions

- Eiffel: pioneering OO language supported pre- and post-conditions on every method.
- C++ and Java support “assert” keyword
  - Programmer defines a statement that must evaluate to true at runtime.
  - Failure of assertion causes exception
- Some languages have debug-only versions, turned off when system considered correct.
  - Dubious trade-off of efficiency for safety.
- Variable roles could provide rigorous basis for fine-granularity assertions in future.

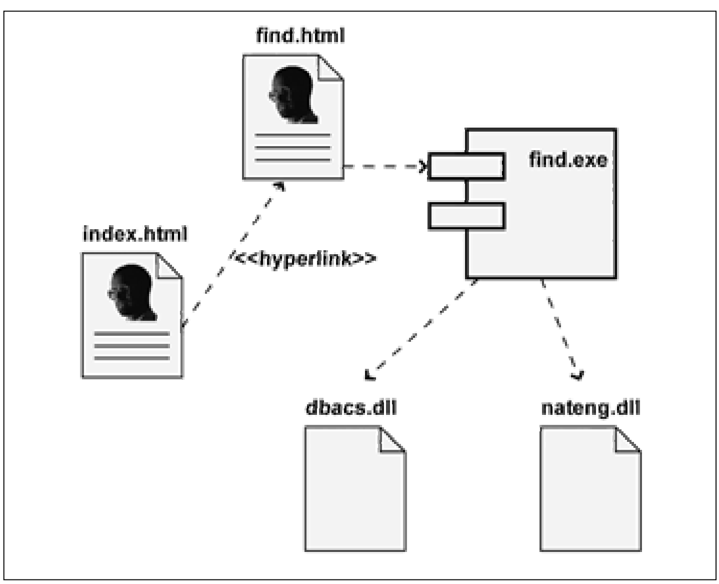
Defensive programming

- Assertions and correctness proofs are useful tools, but not always available.
- Defensive programming includes additional code to help ensure local correctness
  - Treat function interfaces as a contract
- Each function / routine
  - Checks that input parameters meet assumptions
  - Checks output values are valid
- System-wide considerations
  - How to report / record detected bugs
  - *Perhaps* include off-switch for efficiency

More construction

components and reuse

UML Component diagram

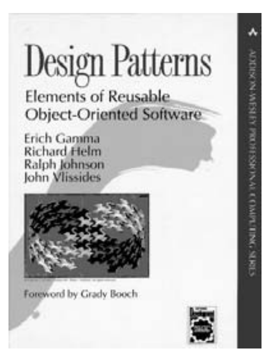


Components and design patterns

- Building applications from existing components is a high priority.
  - In OO design, components are often defined by classes, or packages/libraries of classes
- Inter-class relationships are also important, and can be complex.
  - Some relationships recur in different applications.
  - Design patterns help clarify relationships between classes, and promote reuse of standard approaches to those relationships.

Pioneers – Erich Gamma

- Design Patterns
  - 1995 with the “Gang of Four”
- Don’t reinvent the wheel: educate designers to use known solutions
- Inspired by architect Christopher Alexander
  - A pattern language is a vocabulary of good design.
  - Good design is presented in a “literary” style
  - The most important patterns are internalized.



Pattern structure

- A pattern name.
- The problem addressed by it.
- How it provides a solution:
  - Structures, participants, collaborations.
- Its consequences.
  - Results, trade-offs.

Decorator

- Augments the functionality of an object.
- Decorator object wraps another object.
  - The Decorator has a similar interface.
  - Calls are relayed to the wrapped object ...
  - ... but the Decorator can interpolate additional actions.
- Example: `java.io.BufferedReader`
  - Wraps and augments an unbuffered `Reader` object.

Singleton

- Ensures only a single instance of a class exists.
  - All clients use the same object.
- Constructor is private to prevent external instantiation.
- Single instance obtained via a static `getInstance` method.

Factory method

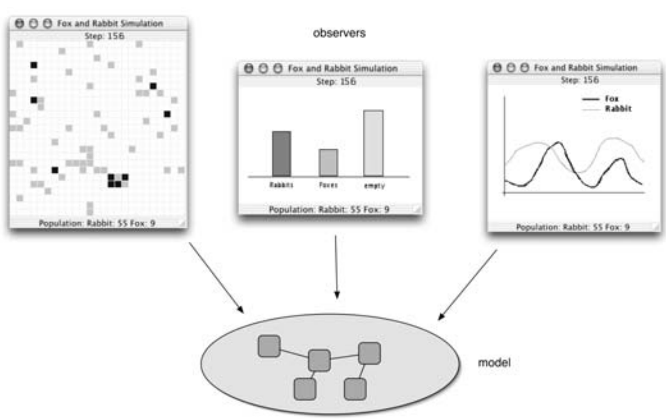
- A creational pattern.
- Clients require an object of a particular interface type or superclass type.
- A factory method is free to return an implementing-class object or subclass object.
- Exact type returned depends on context.
- Example: `iterator` methods of the Collection classes.

Composite

- Class to manage a group of instances that must change state or act together.
- Contains a collection of instances that share a common interface.
- The composite implements the same interface as the individual members.
- Each method implementation in the composite class simply iterates over the contents collection, invoking that method for each member.

Observer

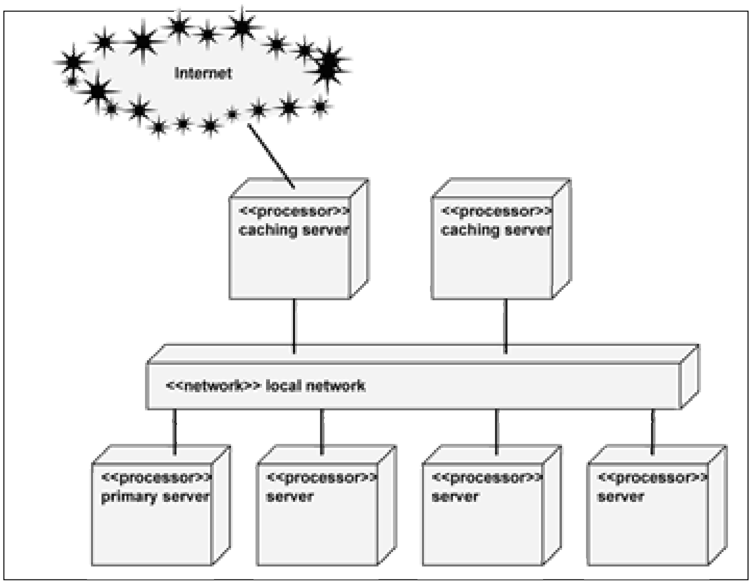
- Separates internal model from views of that model (one-to-many relationship).
- The object-observed notifies all Observers of any state change.



Transition

testing and optimisation

UML Deployment diagram



Pioneers – Michael Fagan

- Software Inspections
  - 1976, IBM
- Approach to design checking, including planning, control and checkpoints.
- Try to find errors in design and code by systematic *walkthrough*
- Work in teams including designer, coder, tester and moderator.

Learning through testing

- A bug is a system's way of telling you that you don't know something (P. Armour)
- Testing searches for the presence of errors.
- Debugging searches for the source of errors.
  - The manifestation of an error may well occur some 'distance' from its source.
  - Need code-reading skills (debugging will often be performed on others' code).
- Techniques and tools exist to support the testing and debugging process.

Test automation

- Good testing is a creative process, but ...
- ... thorough testing is time consuming and repetitive.
- Regression testing involves re-running tests.
- Use of a test rig or test harness can relieve some of the burden.
  - Classes are written to perform the testing.
  - Creativity is then focused in creating these.

Unit testing

- Each unit of an application may be tested.
  - Method, class, module (package in Java).
- Can (should) be done during development.
  - Finding and fixing early lowers development costs (e.g. programmer time).
  - A test suite is built up.
- JUnit helps manage and run tests
  - www.junit.org

Testing fundamentals

- Understand what the unit should do – its *contract*.
  - You look for violations of the contract.
  - Use positive tests (expected to pass) to see whether they don't pass.
  - Use negative tests (expected to fail) to see whether they don't fail.
- Try to test *boundaries*.
  - Zero, one, overflow.
  - Search an empty collection.
  - Add to a full collection.

Manual walkthroughs

- A low-tech approach, relatively underused, but more powerful than appreciated.
- Get away from the computer and 'run' a program by hand.
  - High-level (step) or low-level (step-into) views.
- Monitor and predict object state
  - Object's behaviour is determined by its state.
  - Incorrect behaviour from incorrect state.
  - Tabulate values of all fields.
  - Document state changes after each method call.

Verbal walkthroughs

- Explain to someone else what the code is doing.
  - They might spot the error.
  - The process of explaining might help you to spot it for yourself.
- Group-based processes exist for conducting formal walkthroughs or *inspections*.

Debuggers

- Debuggers are both language- and environment-specific.
- Support breakpoints.
- Step and Step-into controlled execution.
- Call sequence (stack) inspectors.
- Object state – “watch” windows.

Print statements

- The most popular technique.
- No special tools required.
- All programming languages support them.
- Only effective if the right methods are documented.
- Output may be voluminous!
- Turning off and on requires forethought.

Iterative Development

within any design phase or any combination of phases

Prototyping

- Supports early investigation of a system.
  - Early problem identification.
- Incomplete components can be simulated.
  - E.g. always returning a fixed result.
  - Avoid random behavior which is difficult to reproduce.
- Frequent interaction with clients
  - Especially (if feasible) with actual users!

Software changes (or dies)

- There are only two options for software:
  - Either it is continuously maintained
  - or it dies.
- Software that cannot be maintained will be thrown away.
- Not like a novel (written then finished).
- Software is extended, corrected, maintained, ported, adapted…
- The work is done by different people over time (often decades).

Localizing change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.
- Thinking ahead
  - When designing a class, think what changes are likely to be made in the future.
  - Aim to make those changes easy.

Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.
- Code duplication
  - is an indicator of bad design,
  - makes maintenance harder,
  - can lead to introduction of errors during maintenance.

Refactoring and testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Run regression tests before and after refactoring to ensure that nothing has been broken.

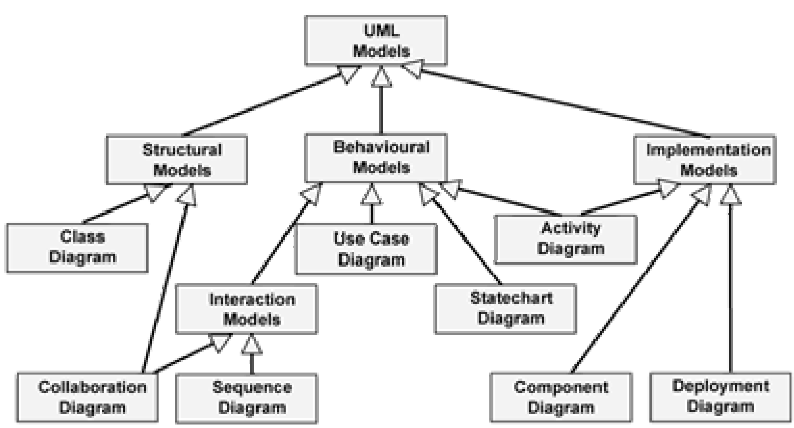
Xtreme Programming' (XP)

- Described in various books by Kent Beck
- An example of an *agile* design methodology
  - Increasingly popular alternative to more "corporate" waterfall/spiral models.
- Reduce uncertainty by getting client/user feedback as soon as possible.
  - Typical team size = two (*pair programming*).
  - Constant series of updates, maybe even daily.
  - Respond to changing requirements and understanding of design by *refactoring*.
- When used on large projects, some evidence of XD (Xtreme Danger)!

Efficiency & Optimisation

- The worst mistakes come from using the wrong algorithm
  - e.g. lab graduate reduced 48 hours to 2 minutes
- Hardware now fast enough to run most code fast enough (assuming sensible algorithms)
  - Optimisation is a waste of your time
- Optimisation is required
  - For *some parts* of extreme applications
  - When pushing hardware envelope
- Cost-effective techniques
  - Check out compiler optimisation flags
  - Profile and hand-optimise bottlenecks

UML review: Modelling for uncertainty



What do engineers know?

- Engineers learn from research, but also invention, design practice, production, and testing.
  - Engineers must know fundamental concepts; criteria and specifications; theoretical tools; quantitative data; practical considerations & design techniques.
- Engineering deals with how things ought to be, science deals with how they are.
  - In science, knowledge is what we *produce*
    - Does it help me understand something?
  - In engineering, knowledge is what we *use*
    - Does it solve a problem?
- Engineers are seldom able to optimize, instead they *satisfice*.
  - They make simplifying models and use these to iterate design, until they achieve a good (not the best) solution.

"To engineer is human" (Petroski)

The requirements for design conflict and cannot be reconciled. All designs for devices are in some degree failures, either because they flout one or another of the requirements or because they are compromises, and compromise implies a degree of failure ... quite specific conflicts are inevitable once requirements for economy are admitted; and conflicts even among the requirements of use are not unknown. It follows that all designs for use are arbitrary. The designer or his client has to choose in what degree and where there shall be failure. … It is quite impossible for any design to be the "logical outcome of the requirements" simple because, the requirements being in conflict, their logical outcome is an impossibility.

David Pye, *The Nature and Aesthetics of Design* (1978).

Technical knowledge is uncertain

- Was the Patriot missile successful in the Gulf war?
  - Even after scientific evaluation of effectiveness in destroying Scuds, the fog of war pervaded technical findings – nobody knows what the “kill ratio” was.
- Who was to blame for the Challenger explosion?
  - NASA were well aware of the effect of cold on O-rings, and debated it (among other risk factors) for years *before* Challenger launch.
  - Technology is never free of risk – systems like the space shuttle should be admired for their complexity, not argued over as if perfectible.
- What did Chernobyl do to Cumbrian sheep?
  - Scientists neglected domain expertise of farmers, who had observed previous contamination from Sellafield, and understood effects of analysis delay on agricultural markets.
  - Public did not recognise uncertainty of science. Instead, when certainty retreats, scientists are accused of political conspiracy.

What do software engineers know?

- Learning by building design models
  - e.g. UML and the unified process
- Inception
  - Usage scenarios, analysis, interaction
- Elaboration
  - Decomposition, modularisation, interfaces
- Construction
  - Structure, coupling, validation
- Transition
  - Testing, optimising, prototyping, iterating